USACERL Interim Report FF-93/07
July 1993
Integrated Persistent Modular Object Representation Translator

US Army Corps
of Engineers
Construction Engineering
Research Laboratories

Research in Persistent Simulation: Development of the Persistent ModSim Object-Oriented Programming Language

by
Charles Herring
Biju Kalathil
Joseph Teo

A general trend in object-oriented programming language development is the addition of features for consistent storage of objects. This capability is implemented through *persistent* programming languages and object-oriented databases. The U.S. Army Construction Engineering Research Laboratories has developed Persistent ModSim, an enhanced version of the ModSim programming language previously developed to provide a general-purpose, object-oriented, process-based simulation language with support for programming large-scale simulations. This report describes the pilot and prototype experiments leading to the development of a robust version of Persistent ModSim. Details of the database class libraries, transient and persistent object allocation, transactions, and compilation management are given. Example programs are provided to show how Persistent Modsim's facilities are used. Directions for future research in this area are outlined.

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN IT IS NO LONGER NEEDED

DO NOT RETURN IT TO THE ORIGINATOR

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1993	Interim

4. TITLE AND SUBTITLE
Research in Persistent Simulation: Development of the Persistent ModSim Object-Oriented Programming Language

5. FUNDING NUMBERS
4A162784
AT41
SE-AV2
Reimb
RP2P69QH12

6. AUTHOR(S)
Charles Herring, Biju Kalathil, and Joseph Teo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Construction Engineering Research Laboratories (USACERL)
P.O. Box 9005
Champaign, IL 61826-9005

8. PERFORMING ORGANIZATION REPORT NUMBER
IR-FF-93/07

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Directorate of Combat Developments (DCD) ATTN: ATSE-CDC-M U.S. Army Engineer School Fort Leonard Wood, MO 65473-5000	Model Improvement and Study Management Agency ATTN: ATSE-CDC-M Operations Analysis Center Fort Leavenworth, KS 66027

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Copies are available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
A general trend in object-oriented programming language development is the addition of features for consistent storage of objects. This capability is implemented through *persistent* programming languages and object-oriented databases. The U.S. Army Construction Engineering Research Laboratories has developed Persistent ModSim, an enhanced version of the ModSim programming language previously developed to provide a general-purpose, object-oriented, process-based simulation language with support for programming large-scale simulations. This report describes the pilot and prototype experiments leading to the development of a robust version of Persistent ModSim. Details of the database class libraries, transient and persistent object allocation, transactions, and compilation management are given. Example programs are provided to show how Persistent Modsim's facilities are used. Directions for future research in this area are outlined.

14. SUBJECT TERMS
object-oriented programming
Persistent ModSim
modeling
simulation

15. NUMBER OF PAGES
56

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

FOREWORD

This study was conducted for the Directorate of Combat Developments (DCD), U.S. Army Engineer School (USAES), under Project 4A162784AT41, "Military Facilities Engineering Technology"; Work Unit SE-AV2, "FAFS-Integrated Persistent Modular Object Representation Translator." The technical monitor was Dave Loental, U.S. Army Engineer School, ATSE-CDC-M. Also described in this report is work done for Model Improvement and Study Management Agency (MISMA), under the reimbursable Work Unit RP2P69QH12, "FAFS-AMIP SimTech ModSim Data Management." The MISMA Technical Monitor was Harry P. Jones, U.S. Army Training and Doctrine Command (TRADOC) Analysis Command.

This research was performed by the Facility Management Division (FF) of the Infrastructure Laboratory (FL), U.S. Army Construction Engineering Research Laboratories (USACERL). Janet H. Spoonamore is Acting Chief, CECER-FF. Dr. Michael J. O'Connor is Chief, CECER-FL.

LTC David J. Rehbein is Commander of USACERL and Dr. L.R. Shaffer is Director.

CONTENTS

RESEARCH IN PERSISTENT SIMULATION: DEVELOPMENT OF THE PERSISTENT MODSIM OBJECT-ORIENTED PROGRAMMING LANGUAGE

1 INTRODUCTION

Background

The Department of Defense (DoD) identifies modeling and simulation (M&S) as a critical technology. This is reflected in the *Army Technology Base Master Plan* (U.S. Army 1992) which lists computer- and electronics-based technology for training and readiness as one of the Army's major science and technology thrusts. In this plan, as in recent DoD and Defense Advanced Research Projects Agency (DARPA) documents (DoD 1992; DARPA 1992), distributed interactive simulation is envisioned as the chief means of training the future force. M&S is further classified as one of the Army's five supporting capabilities. Some applications of M&S as a supporting capability are computer M&S, physical simulation, test and evaluation simulation, and battlefield simulation.

In support of these broad objectives, the Deputy Under Secretary of the Army for Operations Research (DUSA(OR)) established the Army Model and Simulation Management Program (AMSMP) as specified in AR 5-11. AR 5-11 creates a management program with the Army Model Improvement Program (AMIP) and the Simulation Technology Program (SIMTECH) as subordinate entities. AMIP is concerned with the development and management of Army combat models. SIMTECH is concerned with the advancement of simulation technology to further the goals of AMIP. A major M&S technology achievement was the development of the ModSim programming language under SIMTECH sponsorship (Herring 1990). ModSim is a programming language based on Modula-2 and extended with an object type. ModSim's process-based facilities support discrete-event simulation.

USACERL researchers began using ModSim to develop the Force Structure Trade-off Analysis Model (Herring, Wallace, and Whitehurst 1991) as a combat simulation test-bed. During this investigation, ModSim was enhanced to have a consistent mechanism for storing object information. Seamless secondary storage management is a major trend in object-oriented languages. Languages with this feature are called *persistent*. Further research led to pilot and prototype versions of Persistent ModSim, which enables researchers to explore persistent simulation. The addition of consistent object storage, persistence, within the context of the ModSim object-oriented process-based simulation language, provides the user with a "database-centric" view of large-scale simulations. These capabilities dramatically affect the approach to design and implementation of simulations, permitting easy access to functions that would otherwise be available only in a ad hoc manner.

Objective

The objective of the research described in this report is the application of object-oriented database technology to provide a number of enhancements to the ModSim language. The objective of the report is to document the development of Persistent ModSim, and describe its key components and capabilities.

Approach

This report describes the pilot and prototype experiments leading to the development of a robust Persistent ModSim programming language, and describes the Persistent ModSim in detail. This includes

the database class library, transient and persistent object allocation, transactions, and compilation management. The use of these new facilities is illustrated by example programs.

Mode of Technology Transfer

Several journal articles, technical papers, conference presentations, and numerous briefings to Government agencies over the past 2 years have made researchers and developers aware of this research. Under the direction of the DUSA(OR), transfer of Persistent ModSim to several Army agencies has begun. Persistent ModSim is available to any Government agency on request to USACERL.

2 OBJECT-ORIENTED SOFTWARE TECHNOLOGY

Beginning in the mid-1980s, the object-oriented approach to software development replaced the structured approach that flourished during the 1970s. While building on the major concepts of the structured programming languages (e.g., Pascal, C, Ada) such as information hiding, abstract data types, and modularity, the object-oriented approach represents a paradigm shift in software development. Structured programming will remain the arena for software research and development for the remainder of this century, and it is the enabling technology for the integration of many disparate software disciplines. The concept of persistence is central to this integration. This chapter introduces the concepts of object-oriented programming and persistence so that their impact on simulation technology can be appreciated.

The Object-Oriented Paradigm

The object-oriented paradigm is a philosophy for system development. It provides consistent and unifying principles for decision-making at each level of system resolution, including the design, construction, and operation of large software systems. In the context of the evolution of software techniques over the last 30 years, object-oriented programming represents the most advanced embodiment of the goal of software engineering: to produce quality software that precisely fulfills user needs. Quality here means software that is modifiable, efficient, reliable, and understandable (Ross, Goodenough, and Irvine 1975).

True to the nature of a paradigm shift, many of the long-held notions and practices of the structured approach must now be reexamined. For example, the revered top-down approach to systems requirements analysis and design, when seen through "object-colored glasses" seems totally inappropriate. The top-down (structured) approach is characterized by analysis and specification of a system as a set of functions. These functions are derived from the most abstract generalizations about the system. In a process known as "stepwize refinement," these generalized system functions are then broken down into ever more specialized functions until they can be mapped directly into software procedures. The essence of this approach is the focus on function or action. In sharp contrast, the object-oriented approach focuses on the data or objects that make up the system.

Object-oriented software is an organized collection of units that encapsulate both data and the operations or processes that operate on the data. In the structured or functional approach, the data and the operations on them are treated separately. When approaching a problem domain, it is natural to begin by identifying the real-world objects that are part of that domain. The first step is to abstract the attributes and behaviors of these objects into written specifications. Object-oriented software technologies enable programmers to realize the results of such analysis *directly* as software objects and arrange them into software systems that do useful work.

The object-oriented approach is one of modeling—identifying the real-world objects that compose the system. It is well known in control theory that any system (organism) that seeks to control the functions of another system must possess and maintain an internal representation—a model—of that system. It is from the vantage point of a model that analysis proceeds, leading to the understanding and prediction of system behavior. Software systems are models of some real-world enterprises. An accounting system models how a company keeps up with finances; a CAD package lets engineers and architects model buildings; a combat simulation models modern warfare. Thus, object-oriented software technology is ideally suited to support the building of ever more complex software systems.

Object-Oriented Programming Languages

Object-oriented programming is in the phase, inevitable for any new technology, when its concepts are being refined and its terminology clarified. There are currently several approaches to building object-oriented languages. The languages most frequently credited with being the original object-oriented languages are SIMULA (Dahl and Nygaard 1966) and Smalltalk (Goldberg and Robson 1983). Recently, C++ (Stroustrup 1986) and CLOS (Bobrow, et al. 1988) have become popular for new applications development. Despite this diversity, certain features are required for a language to be considered object-oriented. This section briefly reviews these requirements and provides background for the discussion of ModSim in the next chapter.

Two concepts central to object-oriented design and programming are *class* and *object*. In general, a class is a set of things that have the same attributes and behaviors, e.g., cars. In general, an object is a specific instance of a class, e.g., Bill's car. These two concepts have different connotations in software design than they do in programming. In design, the general meanings of class and object, stated above, are used. In implementation (programming languages), class and object take on specific meanings: class is a software unit (module) that encapsulates the attributes and behaviors of some real-world class identified in the design. An object is the instance or realization of an item of a particular class. One of the attractions of the object-oriented approach is the fidelity of the mapping it permits between design and implementation.

As object-oriented languages have evolved, four major characteristics have emerged: information hiding, data abstraction, inheritance, and dynamic binding.

Information hiding was a significant conceptual advance made during the 1970s. It is generally credited to Parnas (1972). According to Parnas, each software unit encapsulates its data and procedures and permits access to its internals only through a well-specified interface. In object-oriented programming, objects communicate by sending messages requesting each other to perform their behaviors. Objects can query each other to find the value of an internal-state variable, but they cannot change this value directly. Information hiding is implemented in programming languages by giving objects the ability to store data structures and operations in a single module. This module provides access through a specification interface and is protected by scope rules from direct manipulation by other objects.

Data abstraction is implemented in a language as the ability to create new types, of which variables are declared. In object-oriented languages, this is the ability to declare an object to be of a certain class. Here the distinction between class and object is again evident. Class corresponds to an abstract data type, and object corresponds to a variable of that abstract type. Data abstraction provides the ability to focus on what is relevant in modeling the problem.

Inheritance is the mechanism by which classes are defined in terms of other classes. Through inheritance, classes can be created that are specializations or extensions of existing classes. The newly created class inherits the data structures and behaviors of its ancestor class. These derived classes are *subclasses*. The process of inheritance can be extended to many generations of subclasses. A familiar example is the class of mammals, with its subclasses (orders, families, etc.) of other mammals representing specializations.

Binding in programming refers to the time at which values are associated with variables. For example, the declaration of a constant, pi=3.1416, is early binding--done at the time the code is written. *Dynamic binding* occurs when values are determined at runtime. Dynamic binding delays the association of data structures and behaviors until runtime. In object-oriented languages, the type of each operand and operation is determined at runtime.

A feature closely related to dynamic binding is *polymorphism*. Polymorphism places the responsibility for correct action on the object. For example, in the mammal analogy given above, all mammals can "move," but they do so in different manners. Some do so on two legs, some on four, and some swim. Thus, the same message sent to different objects will illicite different, but appropriate, behaviors.

A fifth requirement placed on object-oriented languages by some authors is *multiple inheritance* (Meyer 1988). This feature permits classes to be created through inheritance from multiple parent classes.

Persistence and Object Data Management

The concepts of object-oriented programming can be extended naturally to database management. The concept of a persistent object arises naturally from object-oriented programming. A persistent object is an object allocated in secondary storage that is maintained beyond the life of the process that created it. Such an object is retrievable by other, different processes in a well-defined manner. Languages with this feature are referred to as *persistent programming languages*. They are the precursors to *object-oriented database management systems* (ODBMS), which are essentially object-oriented database programming languages. ODBMS applications are written in an existing programming language that has been extended with database management facilities. This illustrates the blending of concepts from programming and database management.

One proposed standard for object-oriented database design has received wide acceptance (Zolonik and Maier 1989). Here the major elements of this model will be stated for use later in developing requirements for persistent simulation. This model is composed of two submodels: a *threshold model* and a *reference model*. The threshold model places requirements on any database system that claims to be object-oriented. These four requirements are:

1. Database functionality
2. Object identity
3. Encapsulation
4. Complex State.

Database Functionality

Database functionality is described in terms of the features found in database systems. These features are provided as follows:

Data Model and Data Manipulation Language. A database has some notion of data structure and a language for manipulating the data. Generally, the data entities are thought of as records and groupings of records are thought of as the structured data provided for in conventional databases. The data manipulation language provides for operations on records and on files composed of records.

Relationships. A database provides for grouping of entities through relationships. In the relational model, these relations are called tables. These relations can be named, and the data-manipulation language can query them. Other database models provide for one-many and many-many relationships, as in the hierarchical and network-type databases.

Permanence. An obvious goal of a database is to provide persistent and stable storage of data. Persistence means that the data is available after the process (program) that created it has terminated. Stability means there is some degree of robustness in cases of failure.

Sharing. Sharing permits data to be accessed by more than a single user, possibly at the same time. This is usually handled through a concurrency-control mechanism.

Arbitrary Size. A database should be able to occupy all of secondary storage. It should not be limited by the processor address range.

Integrity Constraints. Databases provide features for specifying integrity constraints on the entry and manipulation of data. These constraints help ensure correctness. Examples of these constraints are ranges for number fields and required fields. Another form of constraint is referential integrity, that guarantees a reference in one entity does, in fact, reference another entity.

Authorization. Authorization deals with access control and security issues.

Querying. Modern databases provide a query language that is declarative and supports associative access. Declarative means that the user makes a statement as to the goal of a query and the query processor determines the process by which the data is obtained. The query language, especially in relational databases, can retrieve data based on associations, such as relations, among the data entities.

Separate Schema. A database schema is a specification of all the types and of the names of all the objects of those types. Most large database systems maintain the schema in a separate repository. Smaller systems sometimes represent the schema as data within the system itself. The purpose of the schema is to give disparate programs access to the information describing how the database is structured.

Views and Database Administration. Views are interfaces to the database. They are developed for particular classes of users and the operations they perform. A special view of the database is the one provided for database administration. This view provides access to functions not available to user views, such as reorganization, auditing, security, and archiving.

Data Dictionary. A data dictionary is an extension of the schema concept. It provides more information on the data contained in the database such as documentation, input and output format, and relations to other database systems.

Distributed Data Access. Some database systems provide support for distributed data access over multiple machines located at different sites.

Object Identity

Object identity is a major requirement for object-oriented database systems. The identity of an object is independent of the values of the state variables and is invariant for the life of the object. Object identity is the key characteristic that distinguishes object-oriented database systems. Systems that base identity on the value of data variables are called *value-oriented systems*. Relational systems are value-oriented, e.g., key fields are used to create indexes. The network model of database systems is object-oriented.

An object-oriented database system provides for testing the identity of objects. There is the concept of *shallow equal* and *deep equal*. References can be to objects of the same class (shallow equal) or to the same object (deep equal).

Object-oriented systems must address the problem of dangling references resulting from deletion of objects. It should be noted that object-oriented models do not prevent the use of value-oriented access, i.e., keys and relations.

Encapsulation

Encapsulation has the same meaning it has in object-oriented systems. An object encapsulates both its data and the methods that operate on these data.

Complex State

Complex state means that objects can refer to other objects. This system stores pointers to elements of objects and provides means for accessing those objects.

The Reference Model

The reference model subsumes the threshold model and adds the following requirements:

Structured Representation. The reference model goes beyond encapsulation by providing for objects whose state is a compound data structure.

Persistence by Reachability. The reference model requires that objects of any type can be persistent. That is, persistence is orthogonal to type.

Typing of Objects and Variables. Every object instance knows its type and has a method that can respond to a query of its type.

Three Hierarchies. There are three hierarchies in the reference model: specification of types, implementation of representations and methods, and classification of explicit collections of objects.

Polymorphism. Polymorphism is accomplished through dispatching. The actual method chosen at runtime is dependent on the type of the receiving object.

Collections. The reference model provides for aggregate objects such as sets and lists. All set-based queries are against the collections. Collections can be indexed.

Name Spaces. Variables of any type can be persistent, and their names are available through a hierarchical name space.

Queries and Indexes. There must be a query language capable of using the high degree of structure in the database. The language must be aware of the three hierarchies mentioned above.

Relations. Named relations are supported.

Versions. The model requires that versions of an object's state be accessible.

3 MODSIM LANGUAGE OVERVIEW

Object-oriented design and programming are of particular interest to the simulation community. Object-oriented design consists of identifying the real-world objects that make up the system being modeled. Object-oriented programming faithfully represents these real-world objects in a computer simulation. The first object-oriented language was SIMULA 67 (Dahl 1984), which evolved from the simulation language SIMULA I. Object-oriented languages and simulation have been related from the beginning.

In recognition of the trend towards object-oriented programming and its appropriateness for modeling and simulation, the U.S. Army Model Improvement and Studies Management Agency sponsored the development of an object-oriented language for simulation. This new language is called ModSim for *modular simulation* language. The requirements for object-oriented programming languages were established in Chapter 2. ModSim is now discussed with respect to these requirements. Its facilities for supporting discrete-event simulation will be presented also.

ModSim as an Object-Oriented Language

ModSim is a general-purpose, block-structured, object-oriented programming language. The modular structure of the language and its syntax is based on Modula-2 (Wirth 1982). (Modula-2 is a direct descendent of Pascal.) ModSim is not an exact superset of Modula-2, however. For a detailed comparison of the two languages, see Belanger and Rice (1988). For a complete treatment of ModSim, see Mullarney (Mullarney, West, and Belanger 1988). The purpose here is to discuss the object-oriented capabilities of ModSim in light of the criteria presented in Chapter 2. Note: capitalized words in the following discussion are ModSim reserved words.

A ModSim program consists of a `MAIN` module and any number of library modules. Library modules consist of two parts: a `DEFINITION` and an `IMPLEMENTATION`. These modules are stored in separate files and are compiled separately.

A class is defined through the use of the `TYPE` statement as follows:

```
TYPE
    CombatUnit = OBJECT
       Personnel : INTEGER;
       Location,
       Destination : Coordinate;
       Speed : INTEGER;
    END OBJECT;
```

The new class identified as `CombatUnit` is declared similar to a Pascal record structure. One of the attributes, `Coordinate`, is a user-defined type declared elsewhere within the scope of this class.

ModSim has two types of methods: ASK and TELL. CombatUnit is expanded to include methods as follows:

```
TYPE
    CombatUnit = OBJECT
       Personnel : INTEGER;
       Location,
       Destination : Coordinate;
       Speed : INTEGER;
       ASK METHOD Status : INTEGER;
    END OBJECT;
```

CombatUnit now has one method, Status, which returns an INTEGER value. The use of TELL methods will be explained in the next section, "ModSim as a Simulation Language." In ModSim, all class declarations appear in the DEFINITION module, and many classes can be defined within one module.

The IMPLEMENTATION module contains the code for the methods. The details of the CombatUnit class would appear as follows:

```
OBJECT CombatUnit;
    ASK METHOD Status : INTEGER;
    BEGIN
       implementation code
    END METHOD;
END OBJECT;
```

Information hiding is implemented at several levels. The separation of interface specification (DEFINITION) and code implementation (IMPLEMENTATION) permits information hiding and facilitates software reusability through separate compilation. An advantage of separate compilation is that source code for the DEFINITION can be supplied while the IMPLEMENTATION is supplied only in object form. This physically prevents the subverting of the original author's intent and thereby the integrity of the design. The integration of objects into this scheme is quite natural. The class declaration is in the DEFINITION module, and the code for class behaviors is in the IMPLEMENTATION module.

A second level of information hiding lies in the scope rules governing object visibility. The scope rules of ModSim restrict access of attributes and behaviors to those specified in the DEFINITION modules. This provides the data and procedure encapsulation required of objects. The PRIVATE statement provides additional information hiding capability. It is used to restrict access of an objects's data and behaviors to methods within the object itself.

Abstract data types are implemented by the TYPE and VAR statements, just as in Modula-2 and Pascal. ModSim introduces the class declaration in the DEFINITION module, e.g., TYPE CombatUnit = OBJECT. Objects are realized as variables of a class (TYPE) by declaration in the variable (VAR) section of a module.

In the example given above, the class CombatUnit was declared to be of TYPE OBJECT. It is through the TYPE construct that ModSim implements inheritance. Another example of inheritance in ModSim is shown in this code fragment:

```
TYPE
   ArmorUnit = OBJECT( CombatUnit );
   Tanks : INTEGER;
END OBJECT;
```

Notice the use of CombatUnit in the TYPE declaration as a parameter to OBJECT. This specifies that the new class, ArmorUnit, will inherit all the attributes and methods of the class CombatUnit and is further refined by the addition of the Tanks attribute. In ModSim, complex derived classes can be constructed through use of inheritance of multiple-path class hierarchies.

An object of the class ArmorUnit is realized by declaration as a variable of type ArmorUnit. The ModSim standard procedure NEWOBJ is called to allocate storage for object variables. Now consider:

```
VAR
   ArmorPlatoon : ArmorUnit;
.
.
.
NEWOBJ( ArmorPlatoon );
```

The object ArmorPlatoon is now available to the program. Its attributes can be assigned values, and it can interact with other objects.

Other object-related features of ModSim include the ability to: (1) OVERRIDE inherited methods, (2) restrict the scope of attributes and methods through use of the PRIVATE statement, and (3) form groups of related objects (collection classes). ModSim, as with all true object-oriented languages, uses dynamic binding and provides for polymorphic behavior. ModSim also provides for classes to be defined in terms of more than one base type (multiple inheritance). The declaration of a class from two base types is shown in this example:

```
TYPE
  ArmorBattalion = OBJECT ( ArmorUnit, BattalionHeadQuarters );
...
END OBJECT;
```

Classes defined in this way inherit all the attributes and methods of the parent classes. This opens the possibility for ambiguity in attribute and method names. ModSim resolves these conflicts by requiring direct reference to multiply defined attributes and through use of the OVERRIDE statement for ambiguous methods.

ModSim as a Simulation Language

There are two types of discrete-event simulation: *event-oriented* and *process-oriented* (Bratley, Fox, and Schrage 1983). Event-oriented simulations are based on the sequencing of a dynamic event list. Each event has an associated routine that determines when other events are placed on the event list. During the execution of an event routine, simulation time does not advance. The event-list manager advances

time when the next scheduled event occurs. In process-oriented simulation, a process is a sequence of logically related activities ordered in time. The routine implementing the process contains all of its related activities. Each process maintains its own activity list. The system maintains a master activity list containing the next activity from each process's activity list. The process-oriented strategy is more appropriate for object-oriented simulation. ModSim is based on the process model.

Process-oriented simulation in ModSim is supported through the addition of simulation primitives that permit time-elapsing methods. Classes can have multiple, concurrent activities. There are provisions for activities to operate synchronously or asynchronously and to interrupt activities within the same object or in other objects. This section describes the primary ModSim constructs for object-oriented, process-based simulation. These are: simulated time, the `TELL METHOD`, the `WAIT` statement, and the class `TriggerObj`.

Simulated Time

Simulated time is a `REAL` (floating-point) value maintained by the ModSim runtime manager. It is dimensionless and can be used to represent any time resolution desired in a simulation. It is accessed by the function `SimTime()`.

The code fragment below illustrates some of the process-oriented simulation capabilities:

```
FROM SimMod IMPORT SimTime;
   TYPE
      CombatUnit = OBJECT
         Personnel : INTEGER;
         Location,
         Destination : Coordinate;
         Speed : INTEGER;
         ASK METHOD Status : INTEGER;
         TELL METHOD MoveTo( IN : NewDestination : Coordinate);
      END OBJECT;
```

The first line of the example above shows the use of the `IMPORT` statement. The ModSim module `SimMod` contains the procedure `SimTime`. This `IMPORT` statement makes the `DEFINITION` of SimTime visible within the scope of the newly defined `TYPE CombatUnit`.

Also notice the addition of a new method, `MoveTo`, which is a `TELL METHOD`. The operation of `TELL METHODs` in a ModSim program differs from that of `ASK METHODs`. `ASK METHODs` perform like procedure calls in most programming languages. When an `ASK METHOD` is encountered, the program waits for the `ASK` to complete and then executes the next statement. However, when a `TELL METHOD` is encountered, the program does not wait for it to complete; the next statement is executed immediately.

It is use of the `TELL METHOD` combined with the `WAIT` statement that causes simulation time to pass. `WAIT` statements can only appear in `TELL METHODs`, and `TELL METHODs` can contain any number of `WAITs`. The first form of the `WAIT` statement that we will examine is:

```
WAIT DURATION real-valued-expression
   statement sequence
[ ON INTERRUPT
   statement sequence ]
END WAIT;
```

When this form of the WAIT statement is encountered, the statements after the WAIT are executed when the specified simulation time has passed. An optional INTERRUPT clause is provided to permit other objects to stop the WAIT. If the WAIT is interrupted, the statements after the INTERRUPT are executed.

To continue with the example, the TELL METHOD MoveTo would contain a WAIT DURATION statement in which the time to wait is calculated based on the IN parameter Coordinate. Processes can be combined to operate synchronously through the use of another form of the WAIT:

```
WAIT FOR object TO tell-method ( parameter )
   statement sequence
[ ON INTERRUPT
   statement sequence]
END WAIT;
```

Through use of this form of the WAIT, objects can synchronize their activities. This permits the direct expression of logical and physical time dependencies in a natural manner. From the example object we declared above, we could have ArmorBattalion issuing orders to its ArmorPlatoon object:

```
WAIT FOR ArmorPlatoon TO MoveTo (2435)
.
.
.
```

The effect of this statement within a TELL METHOD of ArmorBattalion is to synchronize its activities with those of ArmorPlatoon.

The examples have shown how WAIT statements provide for elapsing simulated time and how they can be used to synchronize activities. There are situations, however, where processes depend on the occurrence of specific conditions. ModSim provides the class TriggerObj to use with the WAIT. TriggerObj allows a method to wait for some arbitrary conditions to be met:

```
WAIT FOR trigger-object TO Fire
.
.
.
```

Trigger objects contain a method, Trigger, that is invoked (fired) by some other method. Its firing permits all methods that are waiting on it to continue.

ModSim provides two constructs to stop methods that have invoked WAITs: Interrupt and TERMINATE. The Interrupt method is called from another method to stop a WAIT statement and invoke its ON INTERRUPT clause. For example, if our ArmorPlatoon's MoveTo method had an ON INTERRUPT clause in its WAIT: Interrupt("MoveTo") would cause the statements in that clause to execute. The TERMINATE statement is called from within a process object's method to stop execution immediately.

Implementation

The ModSim language compiler is implemented as a translator. This translator generates C language source code from ModSim source code. This approach has the advantage of easy porting to other machines. The C code is compiled by the host's native C compiler to object code and then linked to produce an executable program. The needed runtime support libraries are supplied in object form with the ModSim compiler. A compilation manager is also supplied to simplify the task of compilation. ModSim is currently available for IBM-PC compatibles and Sun 3 and Sun 4 computers.

4 PILOT AND PROTOTYPE DEVELOPMENT

A major trend in language design is the integration of database management facilities to address the problems of data-intensive applications. The paradigm shift from structured to object-oriented programming has accelerated this trend. Large-scale simulations are extremely data-intensive. They place great demands on developers to create preprocessing, runtime, and postprocessing environments.

Conventional programming languages lack advanced capabilities for data management. Traditionally, they are limited to simple forms of input and output manipulation. They can be extended with program libraries and interfaced to database management systems. These approaches remain limited and ad hoc because the internal representation of data is different from the stored representation. The data must be converted from input format to internal format and from internal format to output storage format. This problem is known as *impedance mismatch* (Copeland and Maier 1984). The object-oriented languages stimulated interest in providing consistent mechanisms for secondary storage management of internal-format representation of data (objects).

This approach to consistent data management within large-scale simulations motivated experiments with a persistent version of the ModSim simulation language. A pilot version was developed to explore the benefits of persistence and to refine requirements. Based on the success of this pilot version, a more robust prototype was implemented. This chapter describes the design rationale and implementation of these experimental versions of Persistent ModSim.

Pilot Experiment

First, a note on the implementation environment and the limitations it placed on the development of the pilot version of Persistent ModSim. A version of ModSim was used on IBM-compatible microcomputers running MS-DOS.* The purpose was to develop the Force-Structure Trade-off Analysis (FSTAM) combat model (Herring et al. 1991). While the object-oriented data models had been studied and a design for Persistent ModSim had been developed based on them, certain limitations of the implementation environment were known to present problems. Nevertheless, effort was put into refining the language's requirements. At the time, there were no object-oriented database systems running on MS-DOS. This pilot version of Persistent ModSim was developed using a commercial package designed for managing the storage of C language data structures (Database Technologies 1989). There were some limitations on the type of data this package could store. The effect of these limitations will be explained. In this chapter the design and features of this pilot Persistent ModSim will be reviewed in relation to the object-oriented data models. The pilot Persistent ModSim work has been described in detail elsewhere (Herring and Whitehurst 1991). Here it is summarized how the pilot addressed the requirements of the threshold model for object-oriented database systems: database functionality, object identity, encapsulation, and complex state.

Database Functionality

The data entities to be stored in ModSim are objects. The data model developed as a natural extension of ModSim's current object-definition facility. The data or state component of the object was stored in the pilot version. Within the context of the current (compiled) ModSim, it is not feasible to store

*MS-DOS: Microsoft Disk Operating System.

and reload methods, i.e., object code. However, this is the trend in true object-oriented database systems and should be the subject of further consideration for ModSim as these systems develop.

The goal of this pilot development was to provide persistent objects within a simulation language. There are two ways to determine which objects should persist: (1) store all objects created in the simulation or (2) provide additional syntax to let the programmer control which objects persist. Because ModSim was designed for simulation, it was decided that all objects should persist. This approach had several benefits. It simplified both the design and the implementation. It required no language extensions and therefore no changes to existing applications. It also relieved any additional programming burden that might be required by adding syntax.

ModSim produces code to run on one processor at a time. Sharing was not addressed other than to say that, in the context of multiuser and distributed simulations, it is an area of interest for the future. The size of the database files produced should have no inherent limitation other than the amount of secondary storage available.

Having addressed the essential features of database functionality, several of the frequently found features were considered important. Many of the integrity constraints required by database systems are imposed through the user views, e.g., entry forms. The system's constraints were arrived at naturally through the static type checking of the programmer-developed object definitions. The consistent data model approach provided for logical constraints on the manipulation of data. Because ModSim produces code for single-user applications, there was no need for security (authorization) at this time. A separate schema, provided by the underlying data manager, facilitated the storage and dynamic retrieval of class hierarchy information. The concept of views (administration being one) is useful in describing the interfaces to the database management facility employed, particularly the ancillary programs, such as the class hierarchy browser.

Of the less frequent features, report and form management could be developed as ancillary programs to provide for data entry and output. Program generators can be developed for these functions. The schema could be extended to accommodate more of the traditional data dictionary functions to aid in the effort.

Object Identity

Object identity is a major requirement of the threshold model. The addition of object identity would require a change in the workings of the ModSim implementation. Currently, all objects are referenced by memory address calculated as offsets into an array of pointers. Objects do not have identity. ModSim does generate a number for each unique class. This number is stored with the object and used for type checking. It carries no class hierarchy information. The ability to determine the equality of objects is essential for persistent storage. Identity is also necessary for referencing of complex objects. The underlying data management facility that was used had a primitive notion of object identity required for consistent object management. This capability was relied on, but it was used only implicitly.

Encapsulation

The data model was based on ModSim's definition of an object, which naturally provides for encapsulation. The concern was with the storage of data only. There was no need to devise a mechanism to store and retrieve methods (object code) in a compiled language.

Complex Type

The pilot version did not implement a provision for the storage of complex types (objects referenced to other objects). Through the use of object identity available in the underlying data manager, complex objects might have been supported, but with considerable effort. The storage of object references based on memory address requires a dynamic address translation scheme as found in virtual-memory operating systems. This was beyond the level of effort required for the pilot version to be a useful demonstration.

Deficiencies

While the pilot version was very useful as a concept demonstration of persistent simulation, it had limitations for serious application development. Some of these limitations were: the inability to store complex types, lack of query facilities within the language (ancillary programs could be written to query the databases, however), multiuser access, and version control.

Prototype Implementation

Based on the pilot experience, its limitations, and the insights gained, a number of decisions were made about the future direction of the persistent compiler functionality and implementation. Also at this time, Sun SPARCstation workstations running the UNIX operating system were acquired. This opened the way to move the development work into a more robust environment. It also permitted using a commercial object-oriented database manager as the basis for persistence in the prototype. The ObjectStore (Object Design Inc. 1991) DBMS was selected. The following sections give the design rational for the prototype Persistent ModSim compiler, describe its features, and illustrate its application.

Design Rationale

The approach to the prototype version followed the same basic rationale as the pilot version. All objects would be persistent by default. There would be one default database for each application in which all objects would be stored. As in the pilot version, this approach ensured that all existing code would be compatible, and it relieved the programmer from needing any knowledge of persistence. The compilation manager `mscomp` (Belanger et al. 1989) was extended to carry out all the additional instructions needed to compile and link the application program with the database support libraries. Ancillary programs could be written in a consistent manner to operate on any database produced by ModSim simulation.

The ObjectStore database is a very advanced form of persistent database programming system. It provides for full support of many of the desirable features not achievable in the pilot version, most importantly: storage of complex types, multiuser database access, and version control. The prototype made use of complex type storage. However, the researchers lacked sufficient experience with the database manager and the requirements for persistent simulation to commit to implementing multiuser access and version control. Thus, the prototype applications were single-transaction based.

Query Features

One major feature was considered necessary to make the addition of persistence useful and would fit elegantly with the syntax. This was a mechanism for querying persistent objects from the database. This enhancement would facilitate the development of programs to analyze data resulting from the simulation runs of exiting programs written before the addition of persistence. It would also aid in the development of new programs.

Research into introducing querying into persistent programming languages, particularly the work of Agrawal and Gehani (1989), who extended C++ to create O++, was applicable to the prototype. Their construct for querying is straightforward. They demonstrate this notation's power for expressing recursive queries in a relational setting. They accomplish this by introducing an iteration operator. The prototype Persistent ModSim incorporated this query iteration as follows:

FOR *object* IN *class* [SUCHTHAT *expression* [BY *field-name*]] *statement sequence*
END FOR

where *object* is a variable of type OBJECT, and *class* is the name of an object type. The *statement sequence* is executed each time an object is found in the collection whose fields satisfy the SUCHTHAT *expression*. The expression of the BY clause causes the loop iteration to examine stored objects in order of increasing values of the *field-name* specified in the expression.

FOR statements can be nested to express order-independent joins, as follows:

FOR $object_1$ IN $class_1$,
    $object_2$ IN $class_2$,
    .
    .
    .
    $object_n$ IN $class_n$
    [SUCHTHAT *expression*] [BY *field*]
        statement sequence

END FOR

The iterator FORALL is added to permit accessing of class hierarchies:

FORALL *object* IN *class* [SUCHTHAT *expression*] [BY *field-name*] *statement sequence*
END FOR

FORALL spans all classes descending from the class of the specified object. Otherwise it works the same as FOR.

The inclusion of the FOR and FORALL constructs provide the needed query operations at the language definition level.

A Demonstration Application

The U.S. Army TRADOC Analysis Command (TRAC) working with the Los Alamos National Laboratory (LANL) developed the Eagle combat model (Alexander 1991). Eagle is a corps/division-level, deterministic, time-stepped combat model with resolution at battalion level. The approach taken in the development of the Eagle model was to use state-of-the-art expert systems software technology. It was implemented using the Knowledge Engineering Environment (KEE) expert system shell and programmed in Common LISP and Common LISP Object System (CLOS).

Inherent in any ground-based combat model is a representation of terrain. The terrain model chosen for Eagle is an object-oriented terrain model developed at LANL (Powell 1989). This model represents terrain features used by military commanders and provides an object-oriented representation for these features consistent with their use and application in military terrain representation. The goal of this model

is a terrain representation implemented in software data structures. These data structures must correspond to the conceptual objects reasoned on by military terrain analysts and planners and available in the larger context of an expert-system-based combat model. Thus, modeled combat units continually interact with the terrain model moving along mobility corridors, reasoning about avenues of approach, planning future routes, and performing other complex unit behaviors.

As part of a related effort, LANL implemented the Eagle terrain model in ModSim along with a demonstration combat simulation. The basic structure of this program is as follows. Terrain data is read from UNIX "flat" files, one record at a time. The complex object terrain representation is built up of dynamic (transient) ModSim memory objects. A number of "Red" and "Blue" combat units are created and set in motion. These units interact with the terrain and each other for some predetermined amount of simulation time, and the program halts. This program is approximately 30,000 lines of ModSim source code.

This program was selected to test the Persistent ModSim prototype. When the program was compiled, unchanged, with the prototype it ran to completion producing the exact output of the original program.

The Persistent ModSim prototype stores all objects in a single database. A listing of the number, size, and type of objects created is shown in Table 1. The total size of this database is approximately 16 Mbytes. Notice that 168,220 `Point` objects were generated along with 54,102 `LineSegment` objects and 169 terrain aggregate (`Terag`) objects. Persistent ModSim stored the entire object terrain representation complete with all the complex interobject references in a database. This database persisted beyond the run of the program and is available for use by other Persistent ModSim programs. This example will be discussed more later.

Table 1

Objects Created by Persistent ModSim Prototype

Total	Obj Size	Total Size	Type
168220	32	5383040	instance_Points_Point_
54102	80	4328160	instance_LineSegments_LineSegment_
68116	32	2179712	instance_ObjLists_ObjList_
3621	160	579360	instance_MobCorEdges_MobCorEdge_
2198	120	263760	instance_Edges_Edge_
2285	72	164520	instance_RouteGenerators_NTuple_
2274	40	90960	instance_Nodes_Node_
2148	40	85920	instance_Circles_Circle_
1250	52	65000	instance_RawReports_RawReport_
169	232	39208	instance_Terags_Terag_
560	32	17920	instance_SystemNumberPairs_SystemNumberPair_
10	1776	17760	instance_DirectFireMunitions_DirectFireMunition_
88	136	11968	instance_RoadEdges_RoadEdge_
10	1192	11920	instance_DirectFireSystems_DirectFireSystem_
4	1024	4096	instance_ManeuverUnit_ManeuverUnit_
4	968	3872	instance_MilitaryUnit_MilitaryUnit_
33	112	3696	instance_Systems_System_
2	1296	2592	instance_HqUnit_HqUnit_
24	96	2304	instance_IOMod_StreamObj_
20	96	1920	instance_MobCorRoutes_MobCorRoute_
4	408	1632	instance_SensorObjs_SensorObj_
11	128	1408	instance_RiverEdges_RiverEdge_
10	136	1360	instance_RouteGenerators_RouteGenerator_
14	64	896	instance_Rectangles_Rectangle_
1	700	700	instance_Maps_Map_
26	24	624	instance_HqC2s_UnitNumberPairObj_
1	312	312	instance_MainMod_MainObj_
1	296	296	instance_Graphic_GraphObj_
4	40	160	instance_HqC2s_IndicatorObj_
4	40	160	instance_HqC2s_IntelOrderObj_
4	36	144	instance_HqC2s_CollectionRequestObj_
7	20	140	instance_AssocLists_AssocList_
1	132	132	instance_InputObjs_InputObj_
4	32	128	instance_SensorObjs_SensorNumPairObj_
4	28	112	instance_HqC2s_UnitTaskingObj_
1	104	104	instance_InputUnits_InputUnit_
4	24	96	instance_AOIs_AOI_
4	16	64	instance_OAExpert_OAExpert_
2	16	32	instance_PhaseExpert_PhaseExpert_

Total Database Size: 16441344 bytes (16056 Kbytes)

5 PERSISTENT MODSIM

The previous chapter reviewed work on a pilot version of persistent ModSim and on further prototype development. The major decision made in the prototype development effort was the use of the commercial object-oriented database manager ObjectStore. The choice of a full-featured commercial database product for implementing persistence brought with it many design decisions. The major challenge was the tradeoff between preserving the syntax of the existing language definition and providing flexibility in persistent object management. Using the prototype for development of several simulations increased understanding of the requirements for persistent simulation. This understanding led to the implementation of a new version of Persistent ModSim. The approach taken in the new implementation is described here.

Experience showed that programmers need to know and understand the consequences of object persistence. This capability dramatically changes the ways in which programs can be structured. Therefore, programmer's had to be given more access to the underlying database functionality, and this access had to be provided in a consistent manner. The choice was to provide an interface to persistent storage through a database class library.

This solution presents the essential features of database functionality to the user as objects. This approach is common and permits control over how much detail is revealed. The class library represents a simplified model of object database. This approach requires no syntax changes and is consistent with the object paradigm of adding features through classes from which subclasses can inherit. This gives programmers the flexibility to develop further specializations of this model for specific purposes.

This approach introduces two new concepts into the language in the form of extended syntax. They are (1) the specification of persistent allocation of objects and (2) transaction management to permit concurrent access to databases by multiple users. These extensions in no way make existing ModSim code incorrect.

This chapter describes database class approach to implementing a general and robust Persistent ModSim capable of supporting the development of complex, large-scale applications.

Database Classes

This section provides an overview of the database model developed for Persistent ModSim. This model consists of a set of classes representing database functionality to the programmer. The database class library consists of the following classes: `Database`, `Segment`, `DatabaseRoot`, `Collection`, `Cursor`, `Configuration`, and `Workspace`. Each class is implemented as in a separate `DEFINITION` file that can be imported into any ModSim program.

The ModSim database classes have no restrictions on them. They can be inherited from, and their methods can be overridden as necessary. However, there are no corresponding `IMPLEMENTATION` files. The code implementing these classes is written entirely in C. These C implementation files call through to the ObjectStore library functions. (It is a powerful feature of the implementation of ModSim that any existing C library can be easily "objectized" as described above.) The following sections give overviews of each of these database classes. Chapter 6 details these classes and their methods and provides examples of their use.

Database

ModSim variables of the object type (class) `Database` represent databases. Databases can be thought of as files that store objects. In a ModSim program, an instance of a `Database` object is associated with one physical database at any given time. `Database` objects have no fields, only methods. These methods implement the basic notions associated with database operations. The methods of `Database` objects include methods to create, open, close, and delete. There are methods for determining the status of the database and methods associated with the functioning of the other classes in the database library.

Segment

Databases are composed of segments. A segment can be thought of as the smallest unit of memory that is transferred from persistent to transient storage. Every database is created with an initial segment. As objects are stored in the database, additional segments are created automatically. The `Segment` class is provided as a means of clustering groups of objects for performance reasons. Because a segment is the unit of memory transfer, significant performance improvements can be gained by physically collocating related objects. One of the methods of the `Database` class is the creation of segments. This method returns a `Segment` object. The `Segment` class includes methods to determine state information, to control size, and to destroy.

DatabaseRoot

The `DatabaseRoot` class provides for structuring entry points into databases. One of the methods of the class `Database` is the creation of objects of type `DatabaseRoot` in the database on which the method is called. `DatabaseRoot` objects are given string names for later retrieval. These `DatabaseRoot` objects can point to any persistent object stored in the database. Any number of `DatabaseRoot` objects can be created in a given database. Some of the functions of the `DatabaseRoot` object include methods to find a named database root in a database, retrieve the name of a given `DatabaseRoot` object, and to deallocate a `DatabaseRoot` object.

Collection

Objects in the `Collection` class serve to group other objects for convenient manipulation. `Collection` objects can be ordered or unordered, and they can be created in a database, a segment, or a configuration. `Collection` objects have methods to insert and remove objects. There are methods for comparing various properties among collections such as being equal to, being greater than, etc. The `Collection` class also provides methods to query over the field values of the contained objects. This method returns another `Collection` object containing the objects selected by the query.

Cursor

The `Cursor` class is related to the `Collection` class. It provides a means to iterate over the elements of a `Collection` object. An instance of a `Cursor` object is created for a given instance of a `Collection` object. The `Cursor` object has methods to return specific members of the given `Collection` object. Some of the `Cursor` object's methods are first, last, next, etc. The `Cursor` object also has methods to insert objects into its collection. Any number of `Cursor` objects can be instanciated for a given `Collection` object.

Configuration

Instances of the `Configuration` class provide a means to specify groupings of objects that are to be treated as a unit for version control. Object instances of any type can be allocated into a given configuration. This includes objects of type `Configuration`. Thus, the programmer can organize subgroups of related objects with configurations to any level. `Configuration` objects are the unit of both version control and database locking. They can be thought of as long-duration transactions on the database. There can be no conflict when a `Configuration` object is checked out to a given application; other applications can also check out versions of the `Configuration` object. Configurations provide the mechanism to achieve change management in shared environments. Some of the methods of the class `Configuration` include checking out, checking out on a branch, and checking in. As versions of a `Configuration` object are checked out, changed, and checked back in, a version tree is formed that permits users to go back to any previous version. There are methods for traversing the version tree of a particular `Configuration` object to retrieve past versions.

Workspace

The `Workspace` class is related to use of configurations. `Workspace` objects provide a way to structure shared and private access to `Configuration` objects. Calls to the various check-out methods of `Configuration` objects are relative to the current workspace. `Workspace` objects are linked (or nested) hierarchically into a workspace tree. Applications can set the access privileges to parts of this workspace tree to control access (and hence change). There must be a default global workspace. `Workspace` objects are then allocated within the context of this global workspace. Workspaces combined with configurations supply the needed concepts for computer supported collaborative work. The `Workspace` class includes methods to create child workspaces of a parent workspace, to get the parent of given workspace, and to set a `Workspace` object to be the current workspace.

Transient and Persistent Object Allocation

The ModSim built-in function `NEWOBJ` is used for dynamic allocation of transient object instances. This function takes an argument of type `OBJECT`, allocates transient memory for the object instance fields, sets up virtual function tables for its methods, and calls its default initialization method, `ObjInit`. In Persistent ModSim, this function was extended to permit persistent allocation of object instances. Persistent ModSim's `NEWOBJ` function permits the use of an additional parameter to specify where the given instance will be allocated in persistent storage. This additional parameter must be an instance of an object of type `Database`, `Segment`, or `Configuration`. Objects allocated in this manner are persistently allocated in the corresponding physical database, database segment, or configuration within a database (or segment). This approach permits persistent allocation of any object type. This property is known as *persistence orthogonal to type.* It places no restriction on persistent allocation of existing objects or on the construction of new ones. Note that the built-in function for deallocation of transient object instances, `DISPOSEOBJ`, also works for deallocation of persistent objects.

The rules for assignment of transient and persistent objects (pointers) are as follows:

- Transient pointers can point to persistent memory.
- Persistent pointers can point to transient memory.
- All transiently allocated memory is valid only during the process.

- Memory allocated in a database that is assigned to transient locations is valid only during the process.

- Persistent pointers across databases are valid only during the process.

- Both transient and persistent pointers are valid across all transactions.

Transactions and Concurrent Database Access

The concept of a *transaction* is basic to shared database systems. Transactions are necessary to permit multiuser concurrent access to shared data in a database. For applications to read and update the same data consistency, protocols have been developed to overcome problems, such as deadlock, that arbitrate access to shared data.

A transaction is a sequence of program statements that has exclusive control over some shared data. Once an application has achieved control of a portion of a database, other applications must wait for it to finish before they have access to that data. Thus, transactions permit a group of program actions on data to occur without interruption. This is necessary to guarantee integrity of the database system. Note that transaction boundaries within concurrent database applications must be planned wisely. The goal is to permit other applications access to the data. The amount of data locked within a transaction is a prime consideration in multiuser database applications and must be structured appropriately.

The TRANSACTION Statement

A major design requirement for Persistent ModSim was to provide multiuser concurrent access to shared databases. This requires some form of transaction management flexibility within the language. In Persistent ModSim, this is accomplished with the TRANSACTION statement, which accommodates concurrent database access. The Backus-Naur Form (BNF) representation of the TRANSACTION statement is as follows:

```
TransactionStatement =>
   TRANSACTION StatementSequence
   [ON ABORT StatementSequence]
   END TRANSACTION
```

All Persistent ModSim programs that access persistent objects must ensure that such access to these objects in the database takes place within the bounds of a transaction blocked by a TRANSACTION statement. Transactions can be nested to any level required.

The ABORT Statement

An additional built-in function, ABORT, is provided when program logic determines that a transaction must terminate without committing the changes to the database. ABORT terminates the current transaction. Program execution resumes at the next statement after the end of the transaction block (after END TRANSACTION). The ABORT ALL statement aborts all pending transactions in the application program and returns control to the next statement after the end of the top-level transaction statement.

To provide a further level of control when aborting a transaction, the TRANSACTION statement can have an ON ABORT clause. When an ABORT is called within the scope of a TRANSACTION

statement that has an ON ABORT clause, program control is transferred to the section of code starting with the first statement after ON ABORT and continues to the END TRANSACTION of that block. Other, higher-level transactions are not disturbed by aborting a lower-level transaction.

Short-Term and Long-Term Transactions

Within the context of an object-oriented database that provides for version control, the type of transaction processing described above should be considered short term, that is, taking seconds or minutes. The version control provided by Configuration and Workspace objects is intended for long-term transactions. These long-term transactions are the basis for model management within Persistent ModSim. Short-term transactions are for concurrent multiuser access to a single object. Both kinds of transactions are necessary; design requirements and methodology must guide their use.

Two considerations should guide the use of transactions related to version control: (1) only a top-level transaction can be in effect during access to Configuration objects checked out into a workspace, and (2) the method SetCurrent, when called on a Workspace object, will take effect only after the transaction in which it was invoked has ended and a new transaction has begun.

Compilation Management

The ModSim mscomp compilation program manager (CACI 1988) is replaced in Persistent ModSim with pmscomp. To the user, pmscomp works like mscomp, with these differences: (1) The Database library is automatically linked to all applications. Therefore, it is not necessary to specify this library in project files. (2) Two additional configuration options are present in the compilation manager configuration file (pmscomp.cfg): DBPATH and TRY. DBPATH is the database path specification for persistent applications. Each application has certain information associated with it that is stored in this path at compilation time. This option provides a convenient way to structure applications and their databases. Programmers should use this option to segregate disparate database applications, which can greatly improve performance. TRY specifies the number of times a transaction will try to acquire a segment that is locked by another application. The default value of TRY is 10. Applications that experience deadlock problems can be tuned with this option.

6 DATABASE CLASS LIBRARY

In Persistent ModSim, access to persistent object database functionality is achieved via a set of database classes. This chapter provides complete documentation of these classes and provides example programs that illustrate their use. The database classes are `Database`, `DatabaseRoot`, `Collection`, `Cursor`, `Configuration`, `Workspace`, and `Segment`.

The `Database` Class

The class `Database` allows programs to create and manipulate persistent objects. Instances of this class are used as parameters in calls to NEWOBJ to specify where new persistent objects will be allocated. An open count is maintained for each database representing the number of times its `Open` method was called during the current process. When the open count is set to 0, the database is closed. All databases are automatically closed when the program terminates. Object instances of this class need not be persistent. The definition module for the `Database` class is shown below, followed by descriptions of the methods of this class.

```
DEFINITION MODULE Database;

FROM DatabaseRoot IMPORT DatabaseRoot;
FROM Segment IMPORT Segment;

TYPE

    Database =
    OBJECT
      ASK METHOD Close();
      ASK METHOD Create(IN pathname : ARRAY OF CHAR;
                        IN mode : INTEGER;
                        IN if_exists_overwrite : BOOLEAN);
      ASK METHOD CreateRoot(IN name : ARRAY OF CHAR): DatabaseRoot;
      ASK METHOD CreateSegment(): Segment;
      ASK METHOD Destroy();
      ASK METHOD GetHostName(OUT hostName : ARRAY OF CHAR);
      ASK METHOD GetNumberOfRoots(): INTEGER;
      ASK METHOD GetNumberOfSegments(): INTEGER;
      ASK METHOD GetPathName(OUT pathName : ARRAY OF CHAR);
      ASK METHOD IsOpen(): BOOLEAN;
      ASK METHOD Lookup (IN pathname : ARRAY OF CHAR;
                         IN createMode : INTEGER);
      ASK METHOD Of (IN location : REFERENCE;
                     OUT db : Database);
      ASK METHOD Open(IN readOnly : BOOLEAN);
      ASK METHOD Size(): INTEGER ;
    END OBJECT;
END MODULE.
```

`Close` decrements the open count of the database. If the open count is 0, the database is closed. If the open count is greater than 0, the database access (read or read/write) is returned to the previous access mode. If this method is called from within a transaction, the open count is not decremented until the end of the current outermost transaction.

`Create` creates a new database with the specified pathname and mode. The values for mode are the same as those used in the UNIX `chmod` command. If the parameter `if_exists_overwrite` is set to true, a new database is created even if one by that name already exists; otherwise, a runtime error will occur.

`CreateRoot` creates a root in the database with the specified name. This method returns an object of type `DatabaseRoot`.

`CreateSegment` creates a segment in the database and returns an object of type `Segment`.

`Destroy` deletes the database. This method must be called within a transaction.

`GetHostName` returns the host name of the machine on which the database resides.

`GetNumberOfRoots` returns the number of roots retrieved by the current process.

`GetNumberOfSegments` returns the number of segments in the database.

`GetPathName` returns the pathname of the database. This pathname will always begin with a slash (/).

`IsOpen` returns true if the database is open; otherwise, returns false.

`LookUp` associates the database specified in the pathname with an instance of the class `Database`. If the database is not found, a runtime error occurs unless `createMode` is nonzero. If `createMode` is nonzero and the database was not found, a new database is created. Note that this method does not open the database.

`Of` takes a variable of type `OBJECT` and returns the `Database` object it is stored in.

`Open` opens the database associated with the object. If `readOnly` is nonzero, the database is opened for read access only; otherwise, it is open for read/write.

`Size` returns the size of the database in bytes.

The `DatabaseRoot` Class

The `DatabaseRoot` class provides a means for manipulating database entrypoints. `DatabaseRoot` objects provide for named entrypoints for structuring persistent object access. `DatabaseRoot` objects are associated with (point to) persistent objects. These objects can be found by name in the database. The definition module for the `DatabaseRoot`class is shown below, followed by descriptions of its methods.

```
DEFINITION MODULE DatabaseRoot;

TYPE
    DatabaseRoot =
    OBJECT
      ASK METHOD Free();
      ASK METHOD Find(IN name : ARRAY OF CHAR; IN db : REFERENCE);
```

```
      ASK METHOD GetName(OUT name : ARRAY OF CHAR);
      ASK METHOD GetValue() : REFERENCE;
      ASK METHOD SetValue(IN newItem: REFERENCE);
   END OBJECT;
END MODULE.
```

`Free` deletes the persistent, named entrypoint.

`Find` associates the object instance with a specified, named entrypoint in the specified database.

`GetName` returns the name associated with the `DatabaseRoot` object.

`GetValue` returns a pointer to the entrypoint object associated with the `DatabaseRoot` object.

`SetValue` establishes the specified object as the entrypoint object associated with the `DatabaseRoot` object.

The `Collection` Class

The `Collection` class provides for grouping objects, which are called the elements of the collection. An object can be in the collection more than once. The definition module for the `Collection` class is shown below, followed by descriptions of its methods.

```
DEFINITION MODULE Collection;
FROM Database IMPORT Database;
FROM Segment IMPORT Segment;
FROM Configuration IMPORT Configuration;

TYPE

   Collection =
   OBJECT
      ASK METHOD Assign(IN coll : Collection);
      ASK METHOD Contains(IN elem : REFERENCE): BOOLEAN;
      ASK METHOD Count(IN elem : REFERENCE): INTEGER;
      ASK METHOD CreateInDatabase IN db : Database;
         IN Ordered : BOOLEAN);
      ASK METHOD CreateInSegment  (IN seg : Segment;
         IN Ordered : BOOLEAN);
      ASK METHOD CreateInConfiguration (IN cnf: Configuration;
         IN Ordered : BOOLEAN);
      ASK METHOD Delete ();
      ASK METHOD Difference(IN coll : Collection);
      ASK METHOD Empty(): BOOLEAN;
      ASK METHOD EqualTo(IN coll : Collection): BOOLEAN;
      ASK METHOD GreaterThan(IN coll : Collection): BOOLEAN;
      ASK METHOD GreaterThanOrEqualTo(IN coll : Collection):
         BOOLEAN;
      ASK METHOD Insert(IN elem : REFERENCE);
      ASK METHOD InsertFirst(IN elem : REFERENCE);
      ASK METHOD InsertLast(IN elem : REFERENCE);
```

```
    ASK METHOD Intersection(IN coll : Collection);
    ASK METHOD LessThan(IN coll : Collection): BOOLEAN;
    ASK METHOD LessThanOrEqualTo(IN coll : Collection): BOOLEAN;
    ASK METHOD NotEqualTo(IN coll : Collection): BOOLEAN;
    ASK METHOD Only(): REFERENCE;
    ASK METHOD Pick(): REFERENCE;
    ASK METHOD Query (IN elementTypeName : ARRAY OF CHAR;
                      IN query String : ARRAY OF CHAR;
                      IN db : Database;
                      OUT result : Collection);
    ASK METHOD QueryPick (IN elementTypeName : ARRAY OF CHAR;
                          IN queryString : ARRAY OF CHAR;
                          IN db : Database): REFERENCE;
    ASK METHOD Remove(IN elem : REFERENCE);
    ASK METHOD RemoveFirst();
    ASK METHOD RemoveLast();
    ASK METHOD Union(IN coll : Collection);
  END OBJECT;
END MODULE.
```

`Assign` copies the elements of the specified `Collection` object into the collection. This makes the two collections equal.

`Contains` returns true if the specified object is in the collection.

`Count` returns the number of occurrences of the specified object in the collection.

`CreateInDatabase` creates a collection in the specified database. If `Ordered` is true, the collection maintains the elements in the order they are inserted into the collection.

`CreateInSegment` creates a collection in the specified segment. If `Ordered` is true, the collection maintains the elements in the order they are inserted into the collection.

`CreateInConfiguration` creates a collection in the specified configuration. If `Ordered` is true, the collection maintains the elements in the order they are inserted into the collection.

`Delete` removes all the elements from the collection.

`Difference` removes each element from the collection that is in the specified collection.

`Empty` returns true if the collection contains no elements; otherwise, returns false.

`GreaterThan` returns true if the collection is a proper subset of the specified collection.

`GreaterThanOrEqualTo` returns true if collection is a superset of the specified collection.

`Insert` inserts the specified object into the collection.

`InsertFirst` inserts the specified object as the first element of the collection.

`InsertLast` inserts the specified object as the last element of the collection.

`Intersection` modifies the collection to be the intersection of the collection and the specified collection.

`LessThan` returns true if the collection is a proper subset of the specified collection.

`LessThanOrEqualTo` returns true if the collection is a subset of the specified collection.

`NotEqualTo` returns true if the collection is not equal to the specified collection.

`Only` returns the only element of the collection, if it is a singleton set.

`Pick` returns an arbitrary element of the collection.

`Query` returns a `Collection` object created in the specified database, whose elements are of the specified type and meet the membership criteria specified by the query string.

`QueryPick` returns a single element from the `Collection` object, from the specified database, of the specified type, and meeting the membership criteria specified by the query string.

`Remove` removes the specified element from the collection.

`RemoveFirst` removes the first element of a collection.

`RemoveLast` removes the last element of a collection.

`Union` forms the set union of the elements of the `Collection` object with the elements of the specified collection.

The `Cursor` Class

The `Cursor` class provides a way to iterate over objects of the type `Collection`. `Cursor` objects record the state of an iteration over the collection they were created for. The definition module for the `Cursor` class is shown below, followed by descriptions of its methods.

```
DEFINITION MODULE Cursor;
FROM Collection IMPORT Collection;

TYPE
    Cursor =
    OBJECT
       ASK METHOD Copy(IN src: Cursor);
       ASK METHOD CreateForCollection(IN coll: Collection);
       ASK METHOD DeleteSelf();
       ASK METHOD First(): REFERENCE;
       ASK METHOD Last(): REFERENCE;
       ASK METHOD InsertAfter(IN ref: REFERENCE);
       ASK METHOD InsertBefore(IN ref: REFERENCE);
       ASK METHOD More(): BOOLEAN;
       ASK METHOD Next(): REFERENCE;
       ASK METHOD Previous(): REFERENCE;
       ASK METHOD Current(): REFERENCE;
```

```
      ASK METHOD Null(): BOOLEAN;
      ASK METHOD Valid(): BOOLEAN;
   END OBJECT;
END MODULE.
```

`Copy` copies the specified cursor over any exiting cursor state that exists within the current `Cursor` object.

`CreateForCollection` creates a cursor for the specified collection.

`DeleteSelf` deletes the state of the `Cursor` object and breaks any association with its collection.

`First` returns a pointer to the first element of the collection.

`Last` returns the last element of the collection.

`InsertAfter` inserts an object into the collection after the element the cursor is currently positioned on.

`InsertBefore` inserts an object into the collection before the element the cursor is currently positioned on.

`More` returns true if the cursor is not positioned on the last element of the collection; otherwise, it returns false.

`Next` advances the cursor to the next element in the collection and returns a pointer to that element.

`Previous` moves the cursor to the previous element in the collection and returns a pointer to that element.

`Current` returns a pointer to the element the cursor is positioned on.

`Null` returns true if the cursor is valid, that is, if the cursor is positioned on an element of a collection.

`Valid` returns true if the cursor is located at an element of the associated collection.

The `Configuration` Class

The `Configuration` class provides the unit of version control. It serves to group objects that are to be treated as a unit for version-control purposes. Objects are allocated via the `NEWOBJ` function into a `Configuration` object at the time of their creation. Configurations provide a means for long-duration transactions. In all the methods of the `Configuration` class, if the parameter `recursive` is true, the function of that method is applied to all subconfigurations of the given configuration. Note that configurations must be used within the context of workspaces. The definition module of the `Configuration` class is shown below, followed by descriptions of its methods.

```
DEFINITION MODULE Configuration;

FROM Segment IMPORT Segment;
FROM Workspace IMPORT Workspace;
FROM Database IMPORT Database;
```

```
TYPE

    Configuration =
    OBJECT
      ASK METHOD Create (IN db: Database);
      ASK METHOD CreateChild (IN db: Database): Configuration;
      ASK METHOD Checkin (IN recursive : BOOLEAN);
      ASK METHOD Checkout (IN recursive : BOOLEAN);
      ASK METHOD CheckoutBranch (IN branchName : ARRAY OF CHAR;
                                 IN versionName :ARRAY OF CHAR;
                                 IN recursive : BOOLEAN);
      ASK METHOD CreateSegment (OUT seg : Segment);
      ASK METHOD Delete ();
      ASK METHOD DestroyVersion ();
      ASK METHOD Freeze (IN recursive : BOOLEAN);
      ASK METHOD GetName (OUT name : ARRAY OF CHAR);
      ASK METHOD GetNumAlternatives (): INTEGER;
      ASK METHOD GetNumSegments (): INTEGER;
      ASK METHOD GetVersionByName (IN name : ARRAY OF CHAR;
                                   OUT config : Configuration);
      ASK METHOD IsValidObj (): BOOLEAN;
      ASK METHOD Merge (IN that : Configuration;
                        IN recursive : BOOLEAN);
      ASK METHOD NameVersion (IN name : ARRAY OF CHAR);
      ASK METHOD NewVersion (IN recursive : BOOLEAN);
      ASK METHOD Predecessor (OUT config : Configuration): BOOLEAN;
      ASK METHOD Resolve (IN item : REFERENCE): REFERENCE;
      ASK METHOD SameVersion (IN item1: REFERENCE;
                              IN item2 : REFERENCE): BOOLEAN;
      ASK METHOD SetSuccessor (IN s : Configuration);
      ASK METHOD Successor (OUT config : Configuration): BOOLEAN;
      ASK METHOD Use (IN name : ARRAY OF CHAR;
                      IN recursive : BOOLEAN);
      ASK METHOD UseBranch (IN name : ARRAY OF CHAR;
                            IN recursive : BOOLEAN);
    END OBJECT;
END MODULE.
```

`Create` creates a configuration in the specified database, creating a branch containing the newly created configuration. This initial version of the configuration is set to the default for the current workspace.

`CreateChild` creates a child configuration of the configuration in the specified database.

`Checkin` removes the current version of the configuration from the current workspace, puts it into the parent workspace, freezes it, and makes it current for the parent on the branch that contains it.

`Checkout` creates a new version of the configuration and inserts it into the current workspace. If the parameter `recursive` is true, all subconfigurations of the configuration are also checked out.

`CheckoutBranch` creates a new version of the configuration and inserts it into the current workspace, but a new branch is created.

`CreateSegment` adds a segment to the specified configuration for clustering objects.

`Delete` deletes the configuration from the database.

`DestroyVersion` deletes the version from the current workspace.

`Freeze` blocks all subsequent attempts at write access to the configuration.

`GetName` returns the name of the configuration.

`GetNumAlternatives` returns the number of alternative branches of the configuration.

`GetNumSegments` returns the number of segments in the configuration.

`GetVersionByName` retrieves a given version of the configuration by name.

`IsValidObj` returns true if the configuration is valid (not null) object.

`Merge` checks out the specified configuration and calls `SetSuccessor` to set it to the new version.

`NameVersion` adds the specified name to the list of names for this version.

`NewVersion` creates a new version of the configuration and makes it the current version on its branch.

`Predecessor` returns a pointer to the configuration that is the predecessor of the current configuration. This does not change the current configuration in the workspace. This method returns false if there is no predecessor.

`Resolve` takes a pointer to an object in one version of a configuration and returns a pointer to the corresponding version of the same object in another version of the configuration.

`SameVersion` returns true if the two pointers point to versions of the same object.

`SetSuccessor` sets the configuration's successor to the specified configuration.

`Successor` returns a pointer to the successor of the specified configuration.

`Use` makes that version of the specified named configuration into the current version on its branch in the current workspace.

`UseBranch` changes the current branch of the current workspace to be the branch containing the specified named version.

The `Workspace` Class

The `Workspace` class provides a means for using configurations that can be both shared and private. All manipulation of configurations must take place within a current workspace. Configurations are checked in and out of workspaces. The definition module of the `Workspace` class is shown below, followed by descriptions of its methods.

```
DEFINITION MODULE Workspace;

FROM Database IMPORT Database;

TYPE
    Workspace =
    OBJECT
      ASK METHOD CreateChild(IN name : ARRAY OF CHAR;
                             IN db : Database;
                             OUT ws : Workspace);
      ASK METHOD CreateGlobal (IN db : Database;
                               IN name : ARRAY OF CHAR);
      ASK METHOD Delete ();
      ASK METHOD GetName (OUT name : ARRAY OF CHAR);
      ASK METHOD GetParent (OUT parent : Workspace);
      ASK METHOD Of (IN item : REFERENCE;
                     OUT ws : Workspace);
      ASK METHOD Resolve (IN item : REFERENCE): REFERENCE;
      ASK METHOD SetCurrent ();
    END OBJECT;
END MODULE.
```

`CreateChild` creates a child of the current workspace in the specified database. This new child workspace can be referenced by the specified name.

`CreateGlobal` creates a global workspace in the specified database and with the specified name. This is the root workspace from which all child workspaces are rooted.

`Delete` deletes the current workspace from the database.

`GetName` returns the name of the current workspace.

`GetParent` returns a pointer to the parent workspace of the current workspace. This method does not make the parent current.

`Of` returns a pointer to a workspace in which the specified object resides.

`Resolve` returns a pointer to the version of the specified object made visible by the current workspace.

`SetCurrent` sets the workspace to be the current workspace. This takes effect at the beginning of the next transaction.

The `Segment` Class

The `Segment` class provides the units into which databases are divided. Each segment is the atomic unit of transfer from secondary storage to transient memory. Segments can be used directly to cluster objects physically to enhance performance of memory transfers. The definition module for the `Segment` class is shown below, followed by descriptions of its methods.

```
DEFINITION MODULE Segment;

TYPE

   Segment =
   OBJECT
      ASK METHOD DatabaseOf() : REFERENCE;
      ASK METHOD DestroySelf();
      ASK METHOD Of(IN item : ADDRESS; OUT restult : Segment);
      ASK METHOD GetReadWholeSegment() : BOOLEAN;
      ASK METHOD SetReadWholeSegment(IN state : BOOLEAN);
      ASK METHOD GetSize() : INTEGER;
      ASK METHOD SetSize(IN size : INTEGER);
    END OBJECT;
END MODULE.
```

`DatabaseOf` returns a pointer to the `Database` object containing this segment.

`DestroySelf` deletes this segment from the database.

`Of` returns a pointer to the `Segment` object containing the specified object.

`GetReadWholeSegment` returns true when any object contained in the segment is referenced if the segment is set to be read as a whole from the database.

`SetReadWholeSegment` specifies that the segment is read as a whole from the database when any object contained in the segment is referenced. This is the default.

`GetSize` returns the size of the segment in bytes.

`SetSize` sets the segment size to the specified number of bytes. If the segment is larger than the specified number of bytes, there is no effect.

7 EXAMPLE PROGRAMS

This chapter contains several small example programs that illustrate the use of the Persistent ModSim `Database` class library. These are not simulation programs. They simply show the use of the `Database` classes in storing ModSim objects. The program begins by showing how to store a simple object and retrieve it from the database using the `Database, DatabaseRoot, Collection` and `Cursor` classes. Later, it shows how to use `Workspace` and `Configuration` objects to manage access and change in a multiuser environment. In short, this example shows how Persistent ModSim can be used to achieve *model management.*

Using `Database, DatabaseRoot, Collection,` and `Cursor`

The program begins by defining a simple object, `TestObj`, that will be used in the other example programs. The `DEFINITION` and `IMPLEMENTATION` modules are given below.

```
1       DEFINITION MODULE TestObj;
2       TYPE
3          TestObj = OBJECT
4             TestField1: INTEGER;
5             TestField2: ARRAY [0..12] OF CHAR;
6             TestField3: REAL;
7             TestField4: BOOLEAN;
8             ASK METHOD Set(IN f1:INTEGER;
9                            IN f2:ARRAY OF CHAR;
10                           IN f3:REAL;
11                           IN f4:BOOLEAN);
12            ASK METHOD Print;
13         END OBJECT;
14      END MODULE.
```

Lines 1-3 above set up the declaration of the object `TestObj`. Lines 4-7 indicate that `TestObj` has four fields. (Simple ModSim types are used for this object, but Persistent ModSim makes no restriction on the data types that can be stored. Any legal ModSim type, including `REFERENCE` types, can be declared as fields of objects and stored.) Lines 8-12 give the declaration for the two methods of `TestObj`. The `Set` method permits the assignment of values to the fields of `TestObj`. The `Print` method displays these values on the console. The `IMPLEMENTATION` module for `TestObj` is shown below.

```
1       IMPLEMENTATION MODULE TestObj;
2       FROM StrMod IMPORT StrCpy;
3       OBJECT TestObj;
4          ASK METHOD Set (IN f1:INTEGER;
5                          IN f2:ARRAY OF CHAR;
6                          IN f3:REAL;
7                          IN f4:BOOLEAN);
8          BEGIN
9             TestField1:=f1;
10            StrCpy(TestField2,f2);
11            TestField3:=f3;
12            TestField4:=f4;
13         END METHOD;
14         ASK METHOD Print;
```

```
15        BEGIN
16           OUTPUT("TestField1,INTEGER: ",TestField1);
17           OUTPUT("TestField2,ARRAY OF CHAR: ",TestField2);
18           OUTPUT("TestField3,REAL: ",TestField3);
19           IF TestField4
20              OUTPUT("TestField4,BOOLEAN: TRUE");
21           ELSE
22              OUTPUT("TestField4,BOOLEAN: FALSE");
23           END IF;
24        END METHOD;
25     END OBJECT;
26     END MODULE.
```

The IMPLEMENTATION module contains the code implementing the two methods of TestObj (Set and Print). Because TestField2 is a character array, a procedure, StrCpy, is IMPORTed to copy arrays of characters into variables. This is shown in line 2. Lines 4-13 give the implementation for the Set method. This method simply assigns the values of its four parameters to the four fields of TestObj. The Print method is shown in lines 14-24. This method OUTPUTs to the console the values of the fields of TestObj.

Example1

A simple program illustrating the use of the database classes Database, DatabaseRoot, Collection, and Cursor to store and retrieve instances of TestObj is given below. The program, Example1, performs as follows. It begins by making visible (IMPORTing) the needed class definitions and declaring local variables for object instances. Example1 then begins a TRANSACTION, creating a database and a named database root. A collection is created in the database and assigned to the root. Then three instances of TestObj are allocated in the database, values are assigned to their fields (via Set), and they are Inserted into the collection. To illustrate the syntax of a nested transaction, another TRANSACTION is begun. It creates an instance of Cursor, and each object is retrieved from the collection and its values Printed to the console. After all transactions have ended, the database is closed, and the program terminates. The database and the objects allocated into it remain (persist) and can be retrieved by any Persistent ModSim program.

```
1      MAIN MODULE Example1;
2      FROM Database IMPORT Database;
3      FROM DatabaseRoot IMPORT DatabaseRoot;
4      FROM Collection IMPORT Collection;
5      FROM Cursor IMPORT Cursor;
6      FROM TestObj IMPORT TestObj;
7      VAR
8         database : Database;
9         root : DatabaseRoot;
10        collection : Collection;
11        cursor: Cursor;
12        testobj, testobj1, testobj2, testobj3: TestObj;
13     BEGIN
14        TRANSACTION
15           NEWOBJ(database);
16           ASK database TO Create("/ModSim/Example1",664,TRUE);
17           root:= ASK database TO CreateRoot("a_collection");
18           NEWOBJ(collection,database);
19           ASK collection TO CreateInDatabase(database,TRUE);
20           ASK root TO SetValue(collection);
```

```
21          NEWOBJ(testobj1,database);
22          ASK testobj1 TO Set(1,"testobj1",1.0,TRUE);
23          ASK collection TO Insert(testobj1);
24          NEWOBJ(testobj2,database);
25          ASK testobj2 TO Set(2,"testobj2",2.0,FALSE);
26          ASK collection TO Insert(testobj2);
27          NEWOBJ(testobj3,database);
28          ASK testobj3 TO Set(3,"testobj3",3.0,TRUE);
29          ASK collection TO Insert(testobj3);
30          TRANSACTION
31             NEWOBJ(cursor);
32             ASK cursor TO CreateForCollection(collection);
33             testobj:= ASK cursor First();
34             REPEAT
35                ASK testobj TO Print;
36                testobj:= ASK cursor Next();
37             UNTIL ASK cursor Null();
38          END TRANSACTION;
39          ASK database TO Close;
40       END TRANSACTION;
41    END MODULE.
```

Example2

The program Example2, shown below, illustrates retrieval of objects from a database created by another program. Example2 retrieves objects from the database created in Example1. Example2 is similar to Example1 in that it IMPORTs the same class definitions. Note that Example2 does not allocate any persistent objects, but there must nevertheless be transaction boundaries around operations that access persistent data.

```
1     MAIN MODULE Example2;
2     FROM Database IMPORT Database;
3     FROM DatabaseRoot IMPORT DatabaseRoot;
4     FROM Collection IMPORT Collection;
5     FROM Cursor IMPORT Cursor;
6     FROM TestObj IMPORT TestObj;
7     VAR
8        database : Database;
9        root : DatabaseRoot;
10       collection : Collection;
11       cursor : Cursor;
12       testobj : TestObj;
13    BEGIN
14       TRANSACTION
15          NEWOBJ(database);
16          ASK database TO Lookup("/ModSim/Example1",0);
17          ASK database TO Open(FALSE);
18          NEWOBJ(root);
19          ASK root TO Find("a_collection",database);
20          NEWOBJ(collection);
21          collection:=ASK root TO GetValue();
22          NEWOBJ(cursor);
23          ASK cursor TO CreateForCollection(collection);
24          testobj:= ASK cursor First();
```

```
25          REPEAT
26             ASK testobj TO print;
27             testobj:= ASK cursor Next();
28          UNTIL ASK cursor Null();
29       ASK database TO Close;
30       END TRANSACTION;
31    END MODULE.
```

Using Workspace and Configuration

The examples in this section illustrate the use of the Workspace and Configuration classes. The first example defines a simple object, Engine, that contains Piston objects. Many of the database classes are used, and the Configuration class is introduced. The definition module for Engine is shown below.

The Definition Module for Engine

```
1     DEFINITION MODULE Engine;
2     FROM Configuration IMPORT Configuration;
3     FROM Collection IMPORT Collection;
4     FROM Cursor IMPORT Cursor;
5     TYPE
6        Piston = OBJECT
7           PistonNumber: INTEGER;
8           Displacement: INTEGER;
9           ASK METHOD SetPiston(IN number, displacement : INTEGER);
10          ASK METHOD Print;
11       END OBJECT;
12       Engine = OBJECT
13          ASK METHOD InitEngine(IN engconfig: Configuration);
14          ASK METHOD AddPiston(IN piston : Piston);
15          ASK METHOD GetFirstPiston() : Piston;
16          ASK METHOD GetNextPiston() : Piston;
17          ASK METHOD Print;
18          PRIVATE
19          Pistons : Collection;
20          PistonCursor : Cursor;
21       END OBJECT;
22    END MODULE.
```

The definition module above defines the Piston and Engine objects' external interface to other modules that will use them. Lines 1-4 show that this is the definition module for Engine and import the Collection, Cursor, and Configuration object definitions that will be used in this module. The Piston object type is defined in lines 6-11. Piston has two fields: PistonNumber and Displacement. These fields store the piston number in relation to the engine block and the size of the piston in some unit of volume measure, such as cubic inches. Piston has two methods: SetPiston and Print. SetPiston permits the assignment of values to the fields of a Piston object, and Print displays these values on the console. The Engine object is defined in lines 12-21. Engine has five methods and two private fields. The InitEngine method takes as its only parameter a Configuration object. Its purpose is shown below in the discussion of the implementation of Engine. The method AddPiston adds a new piston into the Engine object. The methods GetFirstPiston and GetNextPiston retrieve the pistons from the Engine object. The Print method displays what Engine knows about itself on the console. The two private

fields of Engine are a collection to store the Piston objects and a cursor to iterate over the collection. The implementation for Engine is shown below.

The Implementation Module for Engine

```
1       IMPLEMENTATION MODULE Engine;
2       FROM Configuration IMPORT Configuration;
3       FROM Collection IMPORT Collection;
4       FROM Cursor IMPORT Cursor;
5       TYPE
6        OBJECT Piston;
7           ASK METHOD SetPiston(IN number, displacement:INTEGER);
8           BEGIN
9              PistonNumber:=number;
10             Displacement:=displacement;
11          END METHOD;
12          ASK METHOD Print;
13          BEGIN
14             OUTPUT("Piston: ",PistonNumber,
15                    "Displacement:",Displacement);
16          END METHOD;
17       END OBJECT;

18       OBJECT  Engine;
19          ASK METHOD InitEngine(IN engconfig: Configuration);
20          BEGIN
21             IF Pistons = NILOBJ
22                NEWOBJ(Pistons,engconfig);
23                ASK Pistons TO CreateInConfiguration(engconfig,TRUE);
24             END IF;
25             NEWOBJ(PistonCursor);
26             ASK PistonCursor TO CreateForCollection(Pistons);
27          END METHOD;

28          ASK METHOD AddPiston(IN piston : Piston);
29          BEGIN
30             ASK Pistons TO Insert(piston);
31          END METHOD;

32          ASK METHOD GetFirstPiston() : Piston;
33          BEGIN
34             RETURN ASK PistonCursor First();
35          END METHOD;

36          ASK METHOD GetNextPiston() : Piston;
37          BEGIN
38             RETURN ASK PistonCursor Next();
39          END METHOD;

40          ASK METHOD Print;
41          VAR
42             index, displacement : INTEGER;
43             piston : Piston;
44          BEGIN
```

```
45              OUTPUT("Engine:");
46              piston := ASK PistonCursor First();
47              REPEAT
48                 ASK piston TO Print;
49                 displacement:=displacement +
50                              ASK piston Displacement;
51                 piston := ASK PistonCursor Next();
52              UNTIL ASK PistonCursor Null();
53              OUTPUT("Engine: Displacement:", displacement);
54           END METHOD;
55        END OBJECT;
56       END MODULE.
```

The Piston object is very similar to TestObj of the last section. Its methods store and print its field values. The Engine object is a little more complex but uses the same database classes with one exception, the Configuration class. Engine's InitEngine method takes a Configuration object as a parameter and allocates the Pistons collection into the engconfig object. The implementor of the Engine object has assumed it will be used within a configuration allocated in a database. InitEngine tests whether a Pistons collection already exists (line 21). If not, a call to NEWOBJ with engconfig as a parameter allocates a Pistons collection into the configuration. This method also allocates a transient PistonCursor object for use in accessing the members of the Pistons collection. The methods AddPiston, GetFirstPiston, and GetNextPiston perform operations on the Pistons collection. The Print method uses PistonCursor to iterate over the collection, calling the Print method on each Piston object in the collection. This method also gathers the total displacement for the Engine and displays it on the console. A main module (program), Create, that uses the Engine module is shown below. Create will be used to illustrate use of the Workspace and Configuration classes.

The Create Program

The Create program, shown below, creates a database, a global workspace, and a user workspace. It sets the user workspace to be the current workspace and creates a configuration in that workspace. Then an Engine object along with its eight pistons was allocated into the configuration. The program ends by checking the configuration into the global workspace.

```
1       MAIN MODULE Create;
2       FROM Database IMPORT Database;
3       FROM DatabaseRoot IMPORT DatabaseRoot;
4       FROM Workspace IMPORT Workspace;
5       FROM Configuration IMPORT Configuration;
6       FROM Engine IMPORT Engine, Piston;
7       VAR
8          DB: Database;
9          WSRoot, CFRoot, EngRoot: DatabaseRoot;
10         GlobalWs, UserWs: Workspace;
11         EngineConfig    : Configuration;
12         V8Engine        : Engine;
13         V8Piston        : Piston;
14         index      : INTEGER;
15      BEGIN
16         TRANSACTION
17            NEWOBJ(DB);
18            ASK DB TO Create("/ModSim/Engine",664,TRUE);
19            NEWOBJ(GlobalWs,DB);
```

```
20          ASK GlobalWs TO CreateGlobal(DB,"GlobalWorkSpace");
21          ASK GlobalWs TO SetCurrent();
22          NEWOBJ(UserWs,DB);
23          ASK GlobalWs TO CreateChild("UserWorkSpace",DB,UserWs);
24          WSRoot:=ASK DB TO CreateRoot("UserWSRoot");
25          ASK WSRoot TO SetValue(UserWs);
26          ASK UserWs TO SetCurrent();
27       END TRANSACTION;
28       TRANSACTION
29          NEWOBJ(EngineConfig,DB);
30          ASK EngineConfig TO Create(DB);
31          ASK EngineConfig TO NameVersion("BigV8Engine");
32          CFRoot:=ASK DB TO CreateRoot("EngineConfiguration");
33          ASK CFRoot TO SetValue(EngineConfig);
34          NEWOBJ(V8Engine,EngineConfig);
35          ASK V8Engine TO InitEngine(EngineConfig);
36          FOR index:=1 TO 8
37             NEWOBJ(V8Piston,EngineConfig);
38             ASK V8Piston TO SetPiston(index,50);
39             ASK V8Engine TO AddPiston(V8Piston);
40          END FOR;
41          ASK V8Engine TO Print;
42          EngRoot:=ASK DB TO CreateRoot("EngineRoot");
43          ASK EngRoot TO SetValue(V8Engine);
44          ASK EngineConfig TO Checkin(FALSE);
45       END TRANSACTION;
46       ASK DB TO Close;
47    END MODULE.
```

The use of Database and DatabaseRoot in the Create program should be familiar from Example1 in the previous section. Lines 1-6 establish the interface to other needed modules. Lines 7-14 declare variables to be used in the program. Of interest here are the GlobalWS and UserWS variables, both of type Workspace, and EngineConfig of type Configuration. There are also variables for V8Engine and V8Piston.

One can begin analyzing the Create program by observing its two transactions. The first transaction (lines 16-27) starts by creating the database DB. Line 19 creates the Workspace object GlobalWS in the database. Line 20 calls the method CreateGlobal on the object GlobalWs and gives it the name GlobalWorkSpace. This illustrates that a global workspace must be allocated in an application that uses workspaces. Line 22 allocates the workspace object UserWs in the database, and Line 23 asks the GlobalWs object to make this a child of itself. This establishes UserWs as a decendent workspace of GlobalWs. Lines 24-25 make a root in the database for UserWs. Finally in line 26, UserWS is asked to make itself the current (active) workspace. A transaction boundary must now occur for the action of line 26 to take effect. It is crucial to remember that a call to SetCurrent on a Workspace object takes effect only after the end of the transaction in which it was called. At the end of the first transaction (line 27), UserWs is the current workspace.

The second transaction begins by allocating a Configuration object, EngineConfig, in the database and giving it a name (lines 29-31). Lines 32-33 make a root in the database for EngineConfig. Next, the V8Engine object is allocated into the EngineConfig configuration by a call to the built-in procedure NEWOBJ. This allocates V8Engineinto EnginConfig, which is in the database DB and in the current workspace, UserWs. Lines 36-40 are a FOR loop to allocate eight V8Piston objects into EngineConfig, set their values, and pass them to V8Engine. There, as in the implementation of Engine, they are stored in the Pistons collection. Lines 41-43

tell the V8Engine object to Print and then create a root in the database for this object. Line 44 causes the V8Engine object to be placed into the parent workspace of UserWs which is GlobalWs. The transaction ends, causing the V8Engine object to be written to disk. The database is closed, and the program ends.

The Modify Program

The Modify program now illustrates the capabilities of workspaces and configurations for consistent management of change in an application. This program, shown below, changes the displacement of the V8Engine object.

```
1       MAIN MODULE Modify;
2       FROM Database IMPORT Database;
3       FROM DatabaseRoot IMPORT DatabaseRoot;
4       FROM Workspace IMPORT Workspace;
5       FROM Configuration IMPORT Configuration;
6       FROM Engine IMPORT Engine, Piston;
7       VAR
8
9          DB : Database;
10         WSRoot,CFRoot,EngRoot: DatabaseRoot;
11         GlobalWs,UserWs: Workspace;
12         EngineConfig: Configuration;
13         V8Engine: Engine;
14         V8Piston: Piston;
15         index: INTEGER;
16      BEGIN
17         TRANSACTION
18            NEWOBJ(DB);
19            ASK DB TO Lookup("/ModSim/Engine",664);
20            ASK DB TO Open(FALSE);
21            NEWOBJ(WSRoot);
22            ASK WSRoot TO Find("UserWSRoot",DB);
23            UserWs:=ASK WSRoot TO GetValue();
24            ASK UserWs TO SetCurrent();
25         END TRANSACTION;
26         TRANSACTION
27            NEWOBJ(CFRoot);
28            ASK CFRoot TO Find("EngineConfiguration",DB);
29            EngineConfig:=ASK CFRoot TO GetValue();
30            ASK EngineConfig TO Checkout(FALSE);
31            NEWOBJ(EngRoot);
32            ASK EngRoot TO Find("EngineRoot",DB);
33            V8Engine:=ASK EngRoot TO GetValue();
34            ASK V8Engine TO InitEngine(EngineConfig);
35            OUTPUT("Original Engine");
36            ASK V8Engine TO Print;
37            OUTPUT("Make A Smaller Engine");
38            V8Piston := ASK V8Engine TO GetFirstPiston();
39            index:=1;
40            REPEAT
41               ASK V8Piston TO SetPiston(index,44);
42               V8Piston:=ASK V8Engine GetNextPiston();
43               INC(index);
```

```
44              UNTIL V8Piston = NILOBJ;
45              ASK V8Engine TO Print;
46              ASK EngineConfig TO NameVersion("SmallV8Engine");
47              ASK EngineConfig TO Checkin(FALSE);
48          END TRANSACTION;
49          ASK DB TO Close;
50      END MODULE.
```

The Modify program is identical to the Create program in lines 1-17. At this point, a transaction begins by finding the database created in the Create program (/ModSim/Engine) and opening it for writing. Lines 21-25 allocate a transient Workspace object, UserWs, and use the database root named UserWSRoot to retrieve the persistent workspace object and assign it to UserWs. UserWs is set to be the current workspace, and the transaction ends. Lines 26-30 use the root established in the Create program to find the EngineConfig configuration and check it out of the global workspace into the current workspace, UserWs. Next, the V8Engine object is retrieved by finding its root and asked to perform its InitEngine method (lines 31-34). The program asks the V8Engine object to Print its current state to the console. Now the program changes the displacement of V8Engine. Line 38 retrieves the first Piston object from the collection stored in V8Engine. Lines 39-44 ask each retrieved Piston object to SetPiston to a new, lesser value. V8Engine is then asked to Print its new values. In line 46, a new name (SmallV8Engine) is given to this version of EngineConfiguration. Line 47 checks this new version back into the global workspace, and the transaction and program end.

Modify illustrates how to write a program that checks out a configuration, in this case V8Engine, into a workspace, change the values of its objects and check it back in. There are now two versions of V8Engine stored in the database: BigV8Engine and SmallV8Engine. More versions of V8Engine (or V6Engine) can be created and organized into a version tree. The methods of the Configuration object can be used to move and chose among these versions. CheckOut-Branch can begin a separate development line off the main branch. Configurations have other capabilities. They can be nested arbitrarily, that is, configurations can be created within configurations and maintain their separate changes in a version tree. Workspace provide for multiuser access to configurations in a distributed networked computing environment. Checkout and Checkin can be tuned to the application's pattern of use.

A Note on Simulation

While the examples have made no mention of simulation, it should be obvious that all the modeling power of ModSim is available when using the database classes. The Engine object could be made more complex by adding other components, such as a CrankShaft, CamShaft, Distributor, or CylinderHeads. TELL methods could model the passage of simulation time and use ModSim's synchronization features. In other words, it is possible to make a persistent simulation of a wide range of Engines, to experiment with different configurations, such as Displacement, and to manage these in a consistent, multiuser environment. Interesting intermediate state information about the running Engine objects could be stored in a standard manner that permits the development of general purpose tools for analysis. In short, one can build a model-management strategy for large, complex simulations using Persistent ModSim's database classes.

8 SUMMARY, CURRENT RESEARCH AND FUTURE DIRECTIONS

The ModSim language was developed to support large-scale, general-purpose simulations. Exploring the possible benefits of persistence was a natural next step in this research. This report has explained the pilot and prototype development work demonstrating the usefulness of this technology. This chapter summarizes the chief benefits expected from using a persistent, object-oriented approach to data management for large simulations. Discussions of areas of current and future research follow.

Summary

First, Persistent ModSim can greatly simplify the development of large simulation systems:

- Use of the database class library can significantly reduce the amount of code required to store information for scenario development and output analysis.

- Because this approach relies on a commercial object database, multiuser and distributed data management are inherent.

- The consistency of the database class library should make development of preprocessing and postprocessing environments easier and more general.

- If minimum care is taken to use common data structures, support tools can be developed that are capable of working over a wide range of simulations.

- Change in a multiuser and distributed environment can be managed through versioning and long-term transactions. This is perhaps the greatest benefit. It provides a most convenient approach to model management. Applications being developed to demonstrate this feature in simulations show great promise.

The Eagle terrain demonstration discussed in Chapter 4 is suited to an application of versioning. The original ModSim program begins by reading in flat UNIX files and building the complex object representation of terrain. This complex memory structure is used by the simulated military units. When the program ends, the terrain representation vanishes—there is no mechanism to store the interobject pointers. With Persistent ModSim, the program can be restructured as follows. The code that reads in the flat files can be made into a separate `Create` program whose only job is to initialize the object representation of the terrain in a `Configuration` object in the persistent database. The `Create` program stores the complex memory representation of the object `terrain` in the database. The original program's initialization code is replaced by a few lines that open the database and check out the terrain configuration. The rest of the simulation remains the same.

A number of efficiencies are introduced with this approach:

- The speed of the simulation can be increased greatly. Reading in flat files and establishing complex object representations in memory need to be done only once.

- The code size of the simulation has been reduced by eliminating the initialization code.

- The terrain can now be accessed in a consistent manner for other simulations, and the database browser can be used to explore the terrain directly. Multiple users can access the terrain database over a network.

An integrated development support tool is being developed currently for Persistent ModSim. This includes a graphical browser for ModSim classes and a program editor based on Emacs (Kaplan 1992).

Advanced technology demonstrations of Persistent ModSim have begun in several Army and DOD agencies. There have also been a number of other benefits to simulation technology as a result of this work. Researching the requirements for persistent simulation has contributed to the requirements for the object-oriented database effort (OpenDB) sponsored by DARPA (Herring and Whitehurst 1991). Another direct consequence of the research into Persistent ModSim is the Integrated Systems Language Environment (ISLE) research effort, which is described below.

Future Directions

USACERL researchers apply a wide range of software technologies and require an advanced software engineering environment. Their goal is to develop a new generation of integrated workstation applications to meet needs of engineers and managers in environment, energy, materials, and infrastructure. These software systems will be characterized by model complexity, integration, distributed databases, and the use of artificial intelligence. Currently available software engineering systems do not provide for the integrated application of major software technologies in a consistent manner. USACERL, Army, and DOD researchers need an advanced, open software engineering environment capable of accommodating evolving requirements to produce quality integrated systems.

USACERL is developing ISLE, a software engineering facility that integrates five software technologies into a single programming environment. The goal is to produce a software development environment to support the development of the next generation of integrated applications. These technologies are: (1) object-oriented programming, (2) process-based discrete-event simulation, (3) object databases, (4) knowledge-based programming, and (5) computer-aided software engineering tools.

Object-oriented programming has become the paradigm for current software development and will be the arena for future software research and development for the forseeable future. It provides a formalism for specifying complex system designs and for their realization as software in a straightforward manner.

Process-based discrete-event simulation is another natural consequence of the object-oriented approach. It is an elegant extension to objects that provide for powerful simulation software development that scales up for large applications.

Object databases are the natural extension of object concepts to data storage. They combine the advantages of commercial database systems with the complex data modeling ability of objects. These systems are rapidly becoming the backbone for all computer-aided design (CAD) applications and will become the standard in other areas now dominated by relational technology.

Knowledge-based programming is a software development strategy that grew from research in artificial intelligence. It is distinguished by the separation of the knowledge (logic, specific rules) from the program (interpreter, inference engine) that operates on it. ISLE unites knowledge-based and object-oriented programming in a single programming language.

Computer-aided software engineering tools are necessary to achieve the synergistic integration of the above technologies into an advanced software engineering environment (SEE). This is the essence of ISLE.

ISLE will permit researchers at USACERL, the Army, and DOD to begin developing the next generation of integrated workstation applications that make full use of previously standalone or interfaced software technologies. ISLE supports the integrated use of object-oriented programming, simulation, knowledge-based programming, and databases to address development needs in complex applications such as concurrent engineering. ISLE inherently provides for the development of software architectures in areas as diverse as organizational modeling, construction management, geographic information systems (GIS),

and combat simulation. ISLE opens the door for development of domain-specific methodologies for problem modeling consistent with the underlying software technologies.

The first phase of ISLE research began with the development of the IMPORT/DOME language system. The IMPORT/DOME language is implemented in a unique manner to support ISLE's goals. At the heart of IMPORT/DOME is an object database. On top of this object database is an interface to a generic object-oriented database (GOODB). A class hierarchy, based on this interface models the intermediate forms of the IMPORT/DOME compilation structures. These classes constitute the basis for an SEE. They permit storage in the object database of IMPORT/DOME programs. The following tools are being built based on the SEE classes: a parser that takes IMPORT/DOME source code and stores as intermediate form in database, a code-generator that generates C++ code based on an intermediate-form representation, and a runtime library that supports the execution of compiled C++ code. This work is scheduled for completion by first quarter FY93.

REFERENCES

Agrawal, R., and N. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++," *2nd International Workshop on Database Programming Languages* (Morgan Kaufmann, 1989).

Alexander, Robert, "Eagle Engineer Model Development Process", *Proceedings, 30th Annual U. S. Army Operations Research Symposium* (1991).

Belanger, R., and S. Rice, *ModSim User's Manual* (CACI Products Company, 1988).

Belanger, R. et al., *ModSim: A Language for Object-Oriented Simulation User's Manual*, Deliverable A-010, Contract DABT60-86-C-1382 (CACI Products Company, 1989).

Bobrow, D.G. et al., "Common LISP Object System Specification," *SIGPLAN Notices*, Vol 23 (September 1988).

Bratley, P., B. Fox, and L. Schrage, *A Guide to Simulation* (Springer-Verlag, 1983).

CACI Products Company, *ModSim: A Language for Object-Oriented Simulation, User's Manual* (1988).

Copeland, G., and D. Maier, "Making Smalltalk a Database System," *SIGMOD '84, Proceedings of the Annual Meeting, SIGMOD Record*, Vol 14, No. 2 (1984), pp 316-325.

Dahl, O.J., B. Myrhaug, and K. Nygaard, *SIMULA 67 Common Base Language* (Norwegian Computing Center, Oslo, 1984).

Dahl, O.J., and K. Nygaard, "SIMULA—An Algol-Based Simulation Language," *Communications of the ACM*, Vol 9, No. 9 (1966).

Database Technologies Inc., *C-Data Manager User's Guide and Reference Manual* (Brookline MA, 1989).

Director of Defense Research and Engineering, *DOD Key Technologies Plan* (July 1992).

Director of Defense Research and Engineering, *Defense Science and Technology Strategy* (July 1992).

Goldberg, A., and D. Robson, *Smalltalk-80: The Language and Its Implementation* (Addison-Wesley, 1983).

Herring, C., J. Wallace, A. Whitehurst, and D. Adams, "Design of an Engineer Functional Area Model Using Next-Generation Software Tools and Methodology," *Proceedings, 30th U.S. Army Operations Research Symposium* (1991), p III-89.

Herring, C., "ModSim: A New Object-Oriented Simulation Language," *Object-Oriented Simulation* (Society for Computer Simulation, 1990).

Herring, C., and A. Whitehurst, "Adding Persistence to an Object-Oriented Simulation Language," *Object-Oriented Simulation*, Raimund K. Ege, Ed. (Society for Computer Simulation, 1991).

Herring, C., and A. Whitehurst, "Application Profile: Persistent Simulation," *Position Papers on DARPA/TI Open Object-Oriented Database* (Texas Instruments, Dallas, TX, 13 March 1991).

Herring, C., J. Wallace, and R. Whitehurst, "Design of an Engineer Functional Area Model Using Next-Generation Tools and Methodology," *Proceedings, 30th Army Operations Research Symposium*, Vol III (1991), pp 89-100.

Kaplan, Simon, "Epoch: GNU Emacs for the X Windowing System", Technical Report (University of Illinois, Department of Computer Science, May 1992).

Meyer, B., *Object-Oriented Software Construction* (Prentice Hall, 1988).

Mullarney, A., J. West, R. Belanger, and S. Rice, *ModSim Tutorial* (CACI Products Company, 1988).

Object Design Inc., *ObjectStore Technical Overview Release 1.1* (May 1991).

Parnas, D., "On the Criteria To Be Used in Decomposing Modules," *Communications of the ACM*, Vol 15, No. 2 (1972), pp 1053-58.

Powell, Dennis, "Object Oriented Terrain Analysis," LA-UR-89-3665 (Los Alamos National Laboratory, 1989).

Ross, D., J. Goodenough, and C. Irvine, "Software Engineering Process, Principles, and Goals," *Computer* (May 1975), p 65.

Stroustrup, B., *The C++ Programming Language* (Addison-Wesley, 1986).

U.S. Army, *Army Technology Base Master Plan*, Volume 1 (February 1992).

Wirth, N., *Programming in Modula-2* (Springer-Verlag, 1982).

Zdonik, S., and D. Maier, *Readings in Object-Oriented Databases* (Morgan Kaufmann, 1989).

DISTRIBUTION

Chief of Engineers
ATTN: CEHEC-IM-LH (2)
ATTN: CEHEC-IM-LP (2)
ATTN: CERD-L
ATTN: CEMP

U.S. Army Engineer School 65473
ATTN: ATSE-CDC-M

TRADOC 66027
ATTN: MISMA

Defense Technical Info. Center 22304
ATTN: DTIC-FAB (2)

10
+27
07/93

☆ U.S. GOVERNMENT PRINTING OFFICE: 1993—3510-S/80033